

Continuation Passing Style

Mattox Beckman
beckman@iit.edu

Illinois Institute of Technology

Objectives

§0 Objectives

It is possible to use functions to represent the *control flow* of a program. This technique is called *continuation passing style*. After today's lecture, you should be able to

- explain what CPS is,
- give an example of a programming technique using CPS, and
- transform a simple function from direct style to CPS.

The GCD Program

§1 A Motivating Example

```

1 # let rec gcd a b =
2   match a,b with
3     a,0 -> a
4   | a,b when a<b -> gcd b a
5   | _ -> gcd b (a mod b);;

```

$\text{gcd } 44 \ 12 \Rightarrow \text{gcd } 12 \ 8 \Rightarrow \text{gcd } 8 \ 4 \Rightarrow \text{gcd } 4 \ 0 \Rightarrow 4$

- The running time of this function is roughly $\mathcal{O}(\lg a)$.

GCD of a list

§1 A Motivating Example

```

1 # let rec gcdstar lst =
2   match lst with
3     [] -> 0
4   | x::xs -> gcd x (gcdstar xs);;
5 val gcdstar : int list -> int = <fun>
6 # gcdstar [44;12];;
7 - : int = 4
8 # gcdstar [44;12;80;6];;
9 - : int = 2

```

- Question: What will happen if there is a 1 near the beginning of the sequence?

```

1 # let rec gcdstar lst =
2   match lst with
3     [] -> 0
4     | 1::xs -> 1
5     | x::xs -> gcd x (gcdstar xs);;
6 val gcdstar : int list -> int = <fun>
7 # gcdstar [44;12;80;6];;
8 - : int = 2
9 # gcdstar [44;12;1;80;6];;
10 - : int = 1

```

- This stops the computation, but a lot of work has already been done.

```

1 # 00 let rec gcdstar lst =
2   01   match lst with
3     02     [] -> 0
4     03     | 1::xs -> goto 5
5     04     | x::xs -> gcd x (gcdstar xs);;
6   05 return 1;;

```

- Of course, this is nonsense.

example2.ml

```

1 let gcdstar lst =
2   let rec aux lst =
3     match lst with
4       [] -> 0
5       | x::xs -> gcd x (aux xs) in
6   if (List.for_all (fun x -> x != 1) lst)
7   then aux lst
8   else 1

```

- A function is in *Direct Style* when it return its result back to the caller.
- A *Tail Call* occurs when a function returns the result of another function call without processing it first.
 - This is what is used in accumulator recursion.
- A function is in *Continuation Passing Style* when it passes its result to another function.
 - Instead of returning the result to the caller, we pass it forward to another function.
 - Functions in CPS “never return”.

- A *continuation* is a function into which is passed the result of the current function's computation.

```

1 # let report x = print_int x; print_newline();;
2 val report : int -> unit = <fun>
3 # let plus a b k =
4   k (a + b);;
5 val plus : int -> int -> (int -> 'a) -> 'a = <fun>
6 # plus 20 22 report;;
7 42
8 # plus 20 22 (fun x -> plus 5 x report);;
9 47

```

```

1 gcdstar [44;12;80] R ⇒ aux [44;12;80] R
2 aux [12;80] (fun r1 -> R (gcd 44 r1))
3 aux [80] (fun r2 -> (fun r1 -> R (gcd 44 r1))
4   (gcd 12 r2))
5 aux [] (fun r3 -> (fun r2 -> (fun r1 ->
6   R (gcd 44 r1)) (gcd 12 r2)) (gcd 80 r3))
7 (fun r3 -> (fun r2 -> (fun r1 -> R (gcd 44 r1))
8   (gcd 12 r2)) (gcd 80 r3)) 0
9 (fun r2 -> (fun r1 -> R (gcd 44 r1)) (gcd 12 r2))
10   (gcd 80 0)
11 (fun r2 -> (fun r1 -> R (gcd 44 r1)) (gcd 12 r2)) 80
12 (fun r1 -> R (gcd 44 r1)) (gcd 12 80)
13 (fun r1 -> R (gcd 44 r1)) 4
14 R (gcd 44 4) ⇒ R 4

```

```

1 # let gcdstar lst k =
2   let rec aux lst newk = match lst with
3     [] -> newk 0
4     | 1::xs -> k 1
5     | x::xs -> aux xs
6       (fun res -> newk (gcd x res))
7   in aux lst k;;
8 val gcdstar : int list -> (int -> 'a) -> 'a = <fun>
9 # gcdstar [44;12;80;6] report;;
10 2
11 # gcdstar [44;12;1;80;6] report;;
12 1

```

```

1 gcdstar [44;12;1;80] R ⇒ aux [44;12;1;80] R
2 aux [12;1;80] (fun r1 -> R (gcd 44 r1))
3 aux [1;80] (fun r2 -> (fun r1 -> R (gcd 44 r1))
4   (gcd 12 r2))
5 R 1

```

- In this example, the computation is built up, but when a 1 is encountered, the computation is simply discarded.

Tail Position A subexpression s of expressions e , if it is evaluated, will be taken as the value of e .

- `if (x > 3) then x + 2 else x - 4`
- `let x = 5 in x + 4`
- `f (x * 3)` — no tail position here.

Tail Call A function call that occurs in tail position.

- `if (h x) then h x else (x + g x)`

Available A function call that can be executed by the current expression. The fastest way to be unavailable is to be guarded by an abstraction (anonymous function).

- `if (h x) then f x else (x + g x)`
- `if (h x) then (fun x -> f x) else (x + g x)`

Step 1 Add a continuation argument to any function call.

$$C[\text{let } f \text{ arg} = e] \rightarrow \text{let } f \text{ arg } k = C[e]$$

- The idea is that every function is going to take an extra parameter. “To whom should I tell the result?”

Step 2 A simple expression in tail position should be passed to a continuation instead of returned.

$$C[\text{return } a] \rightarrow k a$$

assuming a is a constant or variable.

- “Simple” = “No available function calls.”

Step 3 To a function call in tail position, pass the current continuation.

$$C[\text{return } f \text{ arg}] \rightarrow C[f \text{ arg } k]$$

- The function “isn’t going to return,” so we need to tell it where to put the result.

Step 4 A function call not in tail position needs to be built into a new continuation. Be sure your new continuation calls the old one if appropriate!

$$C[\text{return } op \ (f \text{ arg})] \rightarrow C[f \text{ arg } (\text{fun } r \rightarrow k(C[op] r))]$$

before

```

1 # let rec foo lst =
2   match lst with
3   | [] -> b
4   | 0::xs -> foo xs
5   | x::xs -> (+) x (foo xs);;
```

after

```

1 # let rec foo lst k = (* rule 1 *)
2   match lst with
3   | [] -> k b (* rule 2 *)
4   | 0::xs -> foo xs k (* rule 3 *)
5   | x::xs -> foo xs (fun r -> k ((+) x r));;
6   (* rule 4 *)
```

```
example5.ml
1 let add a b k = print_string "Add "; k (a + b);;
2 let sub a b k = print_string "Sub "; k (a - b);;
3 let report n = print_string "Answer is: ";
4                 print_int n;
5                 print_newline ();;
6 let idk n k = k n
7
8 type calc = Add of int | Sub of int
```

```
example5.ml
1 let rec eval lst k =
2   match lst with
3   | (Add x) :: xs -> eval xs (fun r -> add r x k)
4   | (Sub x) :: xs -> eval xs (fun r -> sub r x k)
5   | [] -> k 0
6
7 # eval [Add 20; Sub 5; Sub 7; Add 3; Sub 5] report;;
8 Sub
9 Add
10 Sub
11 Sub
12 Add
13 Answer is: 6
```

```
1 # add 3 5 (fun r -> sub r 2 report);;
2 Add
3 Sub
4 Answer is: 6
5 # add 3 5 (fun r k -> sub r 2 k);;
6 Add
7 - : (int -> 'a) -> 'a = <fun>
8 # add 3 5 ((fun k r -> sub r 2 k) report);;
9 Add
10 Sub
11 Answer is: 6
```

Problem: suppose our calculator cannot handle negative numbers.
We need to change the order of the operations somehow.

```
example6.ml
1 let ordereval lst k =
2   let rec aux lst ka ks =
3     match lst with
4     | (Add x) :: xs -> aux xs
5                       (fun r k -> add r x ka k) ks
6     | (Sub x) :: xs -> aux xs
7                       ka (fun r k -> sub r x ks k)
8     | [] -> ka 0 ks k
9   in
10  aux lst idk idk
```

```

1 ordereval [Add 20; Sub 5; Sub 7; Add 3; Sub 5] report
2 Add
3 Add
4 Sub
5 Sub
6 Sub

```

```

1 ordereval [Add 20; Sub 5; Sub 7] report
2 aux [Add 20; Sub 5; Sub 7] idk idk report
3 aux [Sub 5; Sub 7]
4   (fun r1 k1 -> add 20 r1 idk k1) idk report
5 aux [Sub 7]
6   (fun r1 k1 -> add r1 20 idk k1)
7   (fun r2 k2 -> sub r2 5 idk k2) report
8 aux []
9   (fun r1 k1 -> add r1 20 idk k1)
10  (fun r3 k3 -> sub r3 7
11      (fun r2 k2 -> sub r2 5 idk k2) k3)
12  report

```

```

1 aux []
2   (fun r1 k1 -> add r1 20 idk k1)
3   (fun r3 k3 -> sub r3 7
4       (fun r2 k2 -> sub r2 5 idk k2) k3) report
5 (fun r1 k1 -> add r1 20 idk k1) 0
6   (fun r3 k3 -> sub r3 7
7       (fun r2 k2 -> sub r2 5 idk k2) k3) report
8 add 0 20 idk (* remember idk n k = k n *)
9   (fun r3 k3 -> sub r3 7
10      (fun r2 k2 -> sub r2 5 idk k2) k3) report
11 idk 20
12   (fun r3 k3 -> sub r3 7
13      (fun r2 k2 -> sub r2 5 idk k2) k3) report

```

```

1 idk 20
2   (fun r3 k3 -> sub r3 7
3       (fun r2 k2 -> sub r2 5 idk k2) k3) report
4 (fun r3 k3 -> sub r3 7
5   (fun r2 k2 -> sub r2 5 idk k2) k3) 20 report
6 sub 20 7 (fun r2 k2 -> sub r2 5 idk k2) report
7 (fun r2 k2 -> sub r2 5 idk k2) 13 report
8 sub 13 5 idk report
9 idk 8 report
10 report 8

```

Problems**§6 Activity**

- The `gcdstar` example didn't go as far as it could have, because the function `gcd` was left in direct style. Transform `gcd` into CPS, and then write the `gcdstar` function in CPS to use it. It will look very similar to the lecture CPS version of `gcdstar`.
- Suppose now we want to perform multiplications, and we want to do them after we've done all the additions and subtractions. Write the necessary modifications.
- Now suppose we want to do an early abort if we detect a multiply by zero. How do you do that?

Problem 1**§6 Activity**

Here is the code for `gcd` using CPS.

```

1 # let gcd a b k =
2   let rec aux a b =
3     match a,b with
4     | a,0 -> k a
5     | a,b when a<b -> aux b a
6     | _ -> aux b (a mod b);;

```

Problem 1, ctd**§6 Activity**

Here is the code for `gcdstar` that uses it.

```

1 # let gcdstar lst k =
2   let rec aux lst newk = match lst with
3     [] -> newk 0
4     | 1::xs -> k 1
5     | x::xs -> aux xs
6                       (fun res -> gcd x res newk)
7   in aux lst k;;
8 val gcdstar : int list -> (int -> 'a) -> 'a = <fun>

```

Problem 2**§6 Activity**

```

1 let ordereval lst k =
2   let rec aux lst ka ks km =
3     match lst with
4     | (Add x)::xs -> aux xs
5                       (fun r k -> add r x ka k) ks km
6     | (Sub x)::xs -> aux xs
7                       ka (fun r k -> sub r x ks k) km
8     | (Mul x)::xs -> aux xs
9                       ka ks (fun r k -> mul r x km k)
10    | [] -> ka 0 ks km k
11  in
12    aux lst idk idk idk

```

```
1 let ordereval lst k =
2   let rec aux lst ka ks km =
3     match lst with
4     | (Add x)::xs -> aux xs
5                       (fun r k -> add r x ka k) ks km
6     | (Sub x)::xs -> aux xs
7                       ka (fun r k -> sub r x ks k) km
8     | (Mul x)::xs ->
9       if x = 0 then k 0
10      else aux xs
11              ka ks (fun r k -> mul r x km k)
12   | [] -> ka 0 ks km k
13 in
14   aux lst idk idk idk
```